PART I

PATTERN RECOGNITION IN SEQUENCES

CHAPTER 1

LONGEST COMMON SUBSTRINGS, RELATED PROBLEMS AND APPLICATIONS

Maxime Crochemore[1,2]**, Alessio Langiu**[1,3]**, Filippo Mignosi**[3,4]**, Mario Mirisola**[5]

[1]Department of Informatics, King's College London, London, UK

[2]Laboratoire d'Informatique Gaspard-Monge (LIGM), Université Paris-Est, Marne-La-Vallée, France

[3]Dipartimento di Matematica e Informatica, Università degli Studi di Palermo, Palermo, Italy

[4]Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica (DISIM), Università degli Studi dell'Aquila, L'Aquila, Italy

[5]Dipartimento di Biotecnologie Mediche e Forensi (DIBIMEF), Università degli Studi di Palermo, Palermo, Italy.

1.1 INTRODUCTION

One of the most powerful techniques to study the living organisms is to sequence their genomes. From the analysis of the genome one can in fact infer many biochemical features of a living cell or an organism and, focusing on hu-

(Biological Knowledge, Discovery Handbook). By (M. Elloumi, A. Y. Zomaya)

mans, genome analysis can reveal us many of his phenotypic features (exterior manifestation of our genomic determinants) and his genetic susceptibility to a certain number of diseases (consider that at least $6,000$ genetic diseases exist). After obtaining – with two huge collaborative efforts, one public and another one from a private company, at a cost of 3 billion dollars and 300 million, respectively – the first sequencing of the human genome, DNA sequencing has moved fast forward due to huge commercial interests. This acceleration in sequencing methodologies has been so fast that nowadays the cost of the complete exome sequencing (the sequencing of the coding part of the human genome which is around 10% of the entire genome) of an individual will not exceed $2,000$ dollars. Moreover, the recently announced technology provides the whole human genome at the same price. To make a comparison, the fast growing market of cell phones lowered the price of an apparatus with similar performance from 300 dollars in the nineties to 20 nowadays with a 15-fold drop, whilst genome sequencing decreased its cost more than $1,000,000$ fold. The cost effectiveness of new sequencing approaches has tempt many laboratories throughout the world to invest some of their efforts in sequencing project. The result of this blossoming of sequencing projects is the overwhelming sequencing data now available. In fact, the list of fully sequenced organisms grows bigger and bigger at a fast pace. In addition, the recent possibility to sequence the genome from individual cell [27] gives the biologists a tool to compare the physiological as well as the pathological genetic drift (spontaneous accumulation of neutral or pathological mutations overtime) of the different cell line during the life of an individual. Although this effort is surely having high impact on public health, it needs new tools to help data mining and to give significance to these otherwise anonymous strings of nucleotides.

Bioinformatics is of course the major discipline involved in such a task and the design of algorithms capable to compare sequence faster and deeper is of great interest for the life science community. The need for sequence comparisons has many explanations: Multiple alignment of orthologous protein sequences (that is, protein sharing the same function in distinct species) has helped in finding the most relevant part of a protein based on the knowledge that if one or a group of amino acid residue within an orthologue protein are conserved between different organisms this can be explained by their relevance for protein function, whilst amino acid not conserved between different species shouldn't have the same critical role. In other words if evolution didn't find alternatives this means that that one is the best possibility, otherwise we will find different amino acids at that specific position in different species. Indeed, what D. Gusfield [28] claims as "the *first fact* of biological sequence analysis" is that: "In bimolecular sequences (DNA, RNA or amino acid sequences), high sequence similarity usually implies significant functional or structural similarity". This approach has allowed the identification of critical amino acids for specific proteins allowing for example the prediction of the severity of amino-acid substitutions in certain pathologies. In this respect, the possibility to have an algorithm capable to compare orthologous genes from a very

large dataset of species could give some extra information with respect to the actual state of the art.

From the biotechnological point of view, multiple alignment can give important structural insight. For example, the comparison of the enzyme DNA polymerase (a key enzyme in DNA replication and a valuable tool in every life science lab) obtained from non-homeostatic organisms adapted to live in particular ecological niches such as very high temperature, has allowed the identification of amino acid substitutions which confer a more stable structure at non permissive temperatures [40].

Evolutionary studies would also need more efficient algorithms. As an alternative to classical phylogenies of species (the classical tree of life) based on morphological features, molecular biologists currently use DNA sequences to assess when two species diverged during the evolutionary process. Woese and Fox [66] first proposed the use of molecular phylogeny to study prokaryote phylogenies using differences and similarities of the small subunit ribosomal RNA (SSU rRNA). Typically, single gene sequencing is used to this purpose and the gene of choice has been the gene coding for the 16S rRNA [49]. However, phylogenies obtained comparing single genes are rarely consistent with each other: horizontal gene transfer, unrecognized paralogy and highly variable rates of evolution in specific environments are the likely reasons for these inconsistencies [61]. The advent of completely sequenced genomes could allow the construction of more reliable phylogeny since the comparison of the whole genomes is much less affected by the noises described above. To this purpose, the availability of computational method that allow the simultaneous comparison of the complete genome from different living organisms in a reasonable time and with no need of supercomputer is of great interest for the life scientists [57].

One of the theoretical problem borrowed from the stringology field that suit this purpose is the longest common substring (LCS) problem which is meant to identify the substring, i.e. a sequence of contiguous letters in the given document collection, is common to two or more documents. To put it in bioinformatics terms, given a collection of $\mathcal{K}$ genomes, the longest common substrings problem aim is to find the sequences that appear in at least k different genomes, $2 \leq k \leq \mathcal{K}$. Those common parts are high probably conserved sequences of amino acids that deserve a further closer biological investigation. As D. Gusfield [28] say: "the problem of finding (exactly matching) common substrings in a set of distinct strings arises as a subproblem of many heuristics developed in the biological literature to *align* a set of strings. That problem (is) called multiple alignment problem". On the side of evolutionary studies, the k genomes having a such long common sequence are likely to have a common ancestor in the tree of life.

In this chapter, we present a brief review of the historical solutions for the longest common substring problem and some related problems, and an original solution that is easier to implement and it is candidate to be faster in practice using less space than previous solutions. In Section 1.2, we recall

some basic notions from the stringology field and we define the notation used thoroughly in this chapter. In Section 1.3, we present some historical notes of the longest common substring problem and related problems. In Section 1.4, we review the classic solution which uses the suffix tree and the longest common ancestor. Then, we present, in Section 1.5, an original linear solution which uses the classic *union-find* data structure and, in Section 1.6, we present some variants of this solution which are more space efficient.

1.2 PRELIMINARIES

A string is a sequence of zero or more symbols from an alphabet Σ. A string $\mathcal{T}$ of length n is denoted by $\mathcal{T}[1\,..\,n] = \mathcal{T}_1\mathcal{T}_2\ldots\mathcal{T}_n$, where $\mathcal{T}_i \in \Sigma$ for $1 \leq i \leq n$. The length of $\mathcal{T}$ is denoted by $|\mathcal{T}| = n$. ε is the empty (zero-length) string.

A string w is a factor of $\mathcal{T}$ if $\mathcal{T} = uwv$ for $u, v \in \Sigma^*$; in this case, the string w occurs at position $|u|+1$ in $\mathcal{T}$. The factor w is denoted by $\mathcal{T}[|u|+1\,..\,|u|+|w|]$. A prefix (or suffix) of $\mathcal{T}$ is a factor $\mathcal{T}[x\,..\,y]$ such that $x = 1$ ($y = n$), $1 \leq y \leq n$ ($1 \leq x \leq n$). We define the ith suffix as the suffix starting at position i, i.e. $\mathcal{T}[i\,..\,n]$, $1 \leq i \leq n$.

Given a text $\mathcal{T}$ of length n and a pattern $\mathcal{P}$ of length m such that $m \leq n$, $\mathcal{P}$ is said to occur in $\mathcal{T}$ at position i (i.e., exact match) if and only if $\mathcal{P} = \mathcal{T}[i\,..\,i+m-1]$. The position i is said to be an occurrence of $\mathcal{P}$ in $\mathcal{T}$.

In traditional full-text indexing problems one of the basic data structures is the suffix tree (ST) data structure. Another one is the suffix array (SA). A complete description of a suffix tree and the suffix array is beyond the scope of this chapter, and can be found in any textbook on stringology (e.g., [13, 28, 54]). Here we give a very concise definition of those data structures.

The suffix tree $ST_{\mathcal{T}}$ is a compacted trie of all the suffixes of the text $\mathcal{T}$, that is, any suffix $\mathcal{T}[i\,..\,n]$ of $\mathcal{T}$, $1 \leq i \leq n$, can be read on the suffix tree $ST_{\mathcal{T}}$ in a path starting at the root of the tree and ending in the i-th leaf. The number of leaves in such tree is exactly n and, since any internal explicit node is a branching node, the number of internal nodes (non leaf) is strictly less than n. Hence, the total number of nodes is linear in the length of the text, i.e. $O(n)$. Any substring w of $\mathcal{T}$ can be read from the root to an internal node, either explicit or implicit (i.e., a note in between a compacted path represented by a single edge). Auxiliary informations used by many construction algorithm are suffix links between nodes. If u is the node corresponding to the path aw read down inside the tree starting from the root and v is the node corresponding to the path w, then a link from node u to node v is called suffix link. Given two nodes v and u in $ST_{\mathcal{T}}$, the lower common ancestor $LCA(u, v) = z$ is the lower explicit node traversed by both the path to u and v. The string corresponding to the path from the root to z is the longest common prefix between the string of the path to u and the string of the path to v. Notice that ε is prefix of any string, and the root is a common ancestor of any node in the tree. Given a set $\mathcal{K} = \{D_1\,..\,D_K\}$ of documents (strings), the generalized

suffix tree $ST_{\mathcal{K}}$ is the suffix tree of the text obtained by concatenating the documents in $\mathcal{K}$ with a new symbol $\$_k$ appended at the end of each documents, i.e., $ST_{\mathcal{K}} = ST_{\mathcal{T}}$, $\mathcal{T} = D_1\$_1 \ldots D_k\$_k \ldots D_K\$_K$, where every path is truncated after the first dollar.

The suffix tree was introduced by Weiner [65] together with a linear construction algorithm. Other notably (online) linear construction algorithms (also for integer alphabets) are [10, 15, 52, 64]. The pattern matching occurrence query time is optimal, i.e., $O(m + occ)$, where occ is the number of pattern occurrences.

The recent research trend is to build compact or compressed version of the suffix tree [1, 2, 19, 20, 24, 26, 37, 38, 45, 53, 55, 58], basically using a (compressed) suffix array and some auxiliary structures that provide almost all the suffix tree functionality (for instance, top-down and bottom-up traversal, substring retrieval and pattern matching functionalities).

The suffix array $SA_{\mathcal{T}}$ of a text $\mathcal{T}[1 \ldots n]$ is a permutation of $j[1 \ldots n]$ such that $SA_{\mathcal{T}}[i] = j$ if and only if, $\mathcal{T}[j \ldots n]$ is the i-th suffix of $\mathcal{T}$ in (ascending) lexicographic order. The Suffix Array was first introduced in [51], where an $O(n \log n)$ construction algorithm and an $O(m + \log n + occ)$ time pattern matching solution were presented. Later, linear time construction algorithms for the suffix array were presented [36, 39, 42, 56]. The query time was also improved to the optimal $O(m + occ)$ in [2, 19, 20, 37] with the help of another array essentially storing the lengths of longest common prefixes between lexicographically consecutive suffixes.

We remark that the query time of suffix array (and similar other data structures) always contains a hidden $O(\log |\Sigma|)$ factor, where Σ is the underlying alphabet. However, since in most of the cases the size of the alphabet Σ is a constant, the trend in the literature is to omit the $O(\log |\Sigma|)$ factor from the running times. In this chapter we focus on biological sequences where usually the underlying alphabet size is a very small constant (e.g., 4 for DNA/RNA sequences and 20 for protein/amino acid sequences). Finally, we note that there are several linear time suffix tree and array construction methods that works with integer alphabet as well (e.g., [15, 39, 42]).

1.3 HISTORICAL NOTES AND RELATED PROBLEMS

The problem of finding the longest common substring of *two* strings is a classic problem in string analysis. In 1970, D. E. Knuth conjectured that a linear time algorithm for this problem would be impossible (see [28, Section 7.4], [5], [41]). An almost immediate solution using the suffix tree of concatenated strings separated by a symbol colon was available since 1973, when P. Weiner published his famous linear time construction algorithm for suffix trees [65].

Concerning the LCS problem in its full generality, things seem to be a bit more complex. In the introduction of an unpublished manuscript dated 1973, V. Pratt claims a linear time solution to the LCS problem but the claim

doesn't specify whether the problem is settled for a fixed k or for all the values of k. The section where the details were to be presented is not available and was apparently never finished (see references [375] and [376] of the book of D. Gusfield [28] and reference [PR] in [5]).

In 1992, L. Hui gave a linear time solution to the LCS problem that appeared in the Proceedings of the 3rd Symposium on Combinatorial Pattern Matching [34] where he uses a famous constant-time solution to the LCA in trees coupled with the use of generalized suffix trees. The journal version of this result appeared some years later in [35]. It seems that the natural notion of *generalized* suffix tree for a set of strings was firstly introduced in 1992 by D. Gusfield in [29] and used in the same year by L. Hui in order to settle the LCS problem. This notion thus appeared 22 years after the Weiner's construction algorithm of suffix trees, even if it appeared implicitly in Weiner's paper [65] in the case of a two strings set.

An LCS related problem is the Multiple Maximal Exact Matches problem defined in [33]. For this problem, we are given a set of m strings and want to find all $(m+1)$-tuples $(L, i_1, \cdots, i_m)$ such that the substrings of length L starting at $i_1, \cdots, i_m)$ are the same and cannot be extended, i.e., neither $(L+1, i_1, \cdots, i_m)$ nor $(L+1, i_1-1, \cdots, i_m-1)$ represent a common substring of length $L+1$.

In [50] it is considered the LCS problem in the special case when the value k is maximal, i.e. it is equal to the number of documents. This special case is settled in linear time by using matching statistics.

In [12] the LCS problem is settled by using Compact Direct Aciclyc Word Graphs (CDAWGs) instead of suffix trees.

A further generalization of the LCS problem is the k-common repeated substring problem (CRS) that is the following: Given m strings $\mathcal{T}^1, \mathcal{T}^2, \cdots, \mathcal{T}^m$ of total length n and m positive integers $x_1, \cdots, x_m$, for all k with $1 \leq k \leq m$, simultaneously find a longest string ω for which there are at least k strings $\mathcal{T}^{i_1}, \mathcal{T}^{i_2}, \cdots, \mathcal{T}^{i_k}$ $(1 \leq i_1 < i_2 < \cdots < i_k \leq m)$ such that ω occurs at least x_{i_j} times in $\mathcal{T}^{i_j}$, for each j with $1 \leq j \leq k$. For $x_1 = \cdots = x_m = 1$, this problem coincides with the longest common substring (LCS) problem.

In [43], a linear time algorithm is presented that solves the CRS problem for the special case $x_1 = \cdots = x_m = 2$. In other words, the number of times a substring is repeated within the same string is greater than or equal to 2. Lee et al. [44] mention two drawbacks of this algorithm: It is not easy to implement and it is not memory-efficient. In [44] a linear time algorithm is presented that solves the following special case of the k-common repeated substring problem: Given m strings $\mathcal{T}^1, \mathcal{T}^2, \cdots, \mathcal{T}^m$ of total length n and m positive integers $x_1, \cdots, x_m$, for a *fixed* k with $1 \leq k \leq m$, simultaneously find a longest string ω for which there are at least k strings $\mathcal{T}^{i_1}, \mathcal{T}^{i_2}, \cdots, \mathcal{T}^{i_k}$ $(1 \leq i_1 < i_2 < \cdots < i_k \leq m)$ such that ω occurs at least x_{i_j} times in T^{i_j}, for each j with $1 \leq j \leq k$. In [6] it is presented the first O(n) time algorithm for the general CRS problem. The solution in [6], reported also in [54], is based

on a new linear time algorithm for the LCS problem that makes use of the generalized enhanced suffix array.

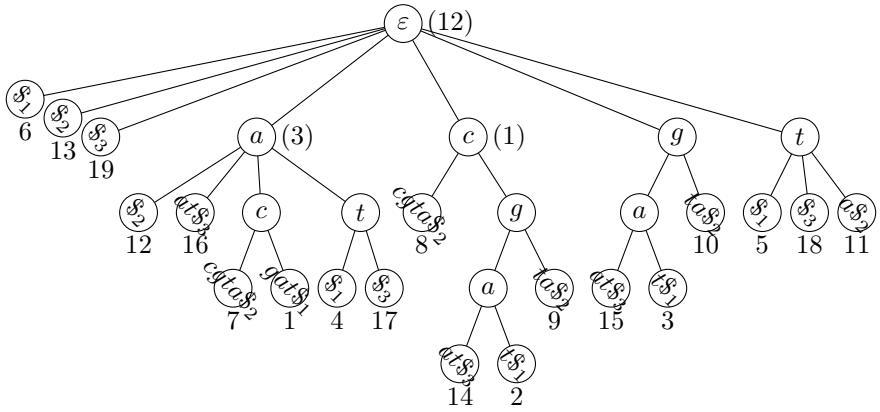
1.4 THE CLASSICAL SOLUTION

Given a set $\mathcal{K}$ of K documents, the Longest Common Substring (LCS) problem (also known in literature as Multiple Common Substring problem and Longest k-Common Substring problem) asks, for any integer $2 \leq k \leq K$, the length of the longest substring which appears in at least k documents. LCS is a well studied problem because of the wide range of applications in Bioinformatics. This problem has been solved by Hui [34, 35] using a generalized suffix tree and a famous constant-time solution for the Lowest Common Ancestor in trees. This solution is reported also in the Gusfield's book [28, Sections 7.6 and 9.7] and it is briefly reviewed in this section.

Let us suppose to have the generalized suffix tree $ST_{\mathcal{K}}$ of the set $\mathcal{K} = \{D_1 \,..\, D_K\}$ of documents. Recall that the generalized suffix tree $ST_{\mathcal{K}}$ is defined as the suffix tree $ST_{\mathcal{T}}$ of the text $\mathcal{T}[1\,..\,n] = D_1\$_1 \,..\, D_k\$_k \,..\, D_K\$_K$. Due to the uniqueness of symbols $\$_k$ and since any internal node in the suffix tree is a branching node and edges are compacted, there are no internal nodes in the tree corresponding to a substring containing a dollar. That is, any substring containing a dollar $\$_k$ corresponds to a leaf or a path ending in the middle of a leaf incoming edge. Hence, we can think to $ST_{\mathcal{K}}$ as a truncated tree, where any path is truncated as soon as a dollar is met. The advantage of the truncated tree is that it does not contain any artificial strings produced by document concatenation. Moreover, the last letter in any path ending on a leaf is a dollar, and, then, it is easy (i.e., a constant time operation) to know, given a leaf i, what document it is a suffix of. Alternatively, we can associate a reference k for the document D_k to each leaf in $ST_{\mathcal{K}}$, where $\$_k$ is the first dollar (left to right) contained in the suffix $\mathcal{T}[i\,..\,n]$.

Now, let us suppose to associate to any internal node v the value $C(v)$ that is the counter of how many different documents are associated with the leaves in the subtree rooted in v. It is easy to prove that one can populate an array L$[k]$, $1 \leq k \leq K$, accumulating the lengths of the longest substring common to exactly k documents while visiting such augmented tree via a depth-first traversal. (In the meantime, by using a simple book keeping strategy, we can store a reference to one of such substrings.) A further scan of the array L will produce the correct $LCS[k]$ array, $2 \leq k \leq K$, where $LCS[k]$ is the length of the longest substring which appears in at least k different documents.

Now the point is how to compute C values. This is a classic problem on trees: The Color-Set-Size (CSS) problem, also known as Colored Sub-Tree problem. L. Hui presented in his paper [34] a solution for the CSS problem and he applied such solution to the generalized suffix tree of the document collection $\mathcal{K}$, in order to settle the LCS problem, by assigning different colors to leaves corresponding to different documents.



List 1	6	1	4	2	3	5	
List 2	13	12	7	8	9	10	11
List 3	19	16	17	14	15	18	

Figure 1.1 Generalized suffix tree of the set $\mathcal{K} = \{acgat, accgta, cgaat\}$ illustrated together with the $K = |\mathcal{K}| = 3$ lists of leaves of a same (document) color. Numbers between parenthesis are counter d values. For instance, if we call v the node of path a, since $\text{LCA}(1, 4) = \text{LCA}(12, 7) = \text{LCA}(16, 17) = v$, then $d(v) = 3$.

Given a tree T with colored leaves, the Color-Set-Size (CSS) problem asks, for each node, the number of different colors in the tree. The classical solution for the CSS problem uses the constant time LCA query for two nodes. Essentially, one computes the desired C value as the number of leaves in the rooted subtree minus the number of duplicated documents (or colors).

He builds K lists containing pointers to all the leaves associated with the k-th document (or color) as they appear at the bottom of the tree in left to right order (that is, K lists of lexicographically ordered leaves of the same color). Obviously, the sum of the length of all the lists is equal to n. After preparing in linear time the generalized suffix tree for constant-time lowest common ancestor (LCA) queries [9, 25, 32, 60], LCA queries are conducted on the leaf lists. In details, for any $1 \leq k \leq K$, let compute the LCA of all the adjacent leaves in the k-th list, i.e., an element and its successor in the list. Let u, v be two of such adjacent leaves and $\text{LCA}(u, v) = z$ be the lowest common ancestor node of u and v.

A counter d at each node z that is the LCA of two consecutive leaves is incremented; Fig. 1.1 shows an example. For any node v, the sum of the counters $d(u)$ of all nodes in the subtree rooted in v is called $D(v)$. It is the number of occurrences of duplicated colors in such subtree and it can be computed via a post-order traversal of the tree. In this way, the number $C(v)$ of distinct colors is computed, for each node v of the tree, as the difference $C(v) = S(v) - D(v)$ between the total number $S(v)$ of leaves in the rooted subtree and the number of duplicated colors $D(v) = \sum_u d(u)$, $u \in \{$subtree

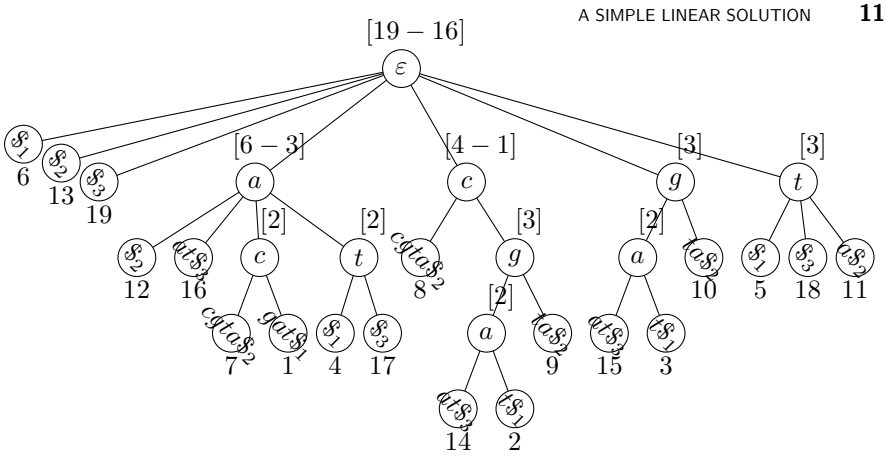


Figure 1.2 Generalized suffix tree of the set $\mathcal{K} = \{acgat, accgta, cgaat\}$ illustrated. $C(v)$ value of any internal node v is reported within square brackets.

rooted in $v\}$. See Fig. 1.2 for an illustration. Notice that $S(v)$ is a standard operation on trees and it can be easily computed within a post-order tree traversal.

In next section, we present a new solution for the CSS problem that uses a well known, fast and space efficient *union-find* data structure instead of the LCA queries used in the classical solution. Even if the time complexity does not get asymptotically improved, the *union-find* based solution has some advantages which make it simple, faste, and easily extendible to more space efficient variants.

1.5 A SIMPLE LINEAR SOLUTION

In previous section we reviewed the classic solution for the longest common substring problem. It essentially reduces LCS problem to the Color-Set-Size problem on the generalized suffix tree of a document collection. Maintaining this approach, we show a simple solution for the CSS problem that turns out to be a simple solution for the LCS problem, as well.

Those two problems are strictly related. Therefore, as in the classical solution, if we build the generalized suffix tree $ST_{\mathcal{K}}$ of a set $\mathcal{K}$ of documents and we color any leaf by assigning a different color to any document, the LCS problem can be solved with a simple traverse of $ST_{\mathcal{K}}$ after that $ST_{\mathcal{K}}$ has been preprocessed to solve the CSS problem.

The CSS solution presented by L. Hui involves the preprocessing of the generalized suffix tree in order to answer efficiently (in constant time) to the Lowest Common Ancestor query of two given nodes.

In this section, we present a new simple solution for the CSS problem, which uses a classical *union-find* data structure. The classic *union-find* data structures for disjoint sets have myriads of practical applications (see for instance

Algorithm 0 Recursive processing the tree T to compute the C values.

```
1: function ColorSetSize(v)
2:     set ← ∅
3:     if v is a leaf then
4:         union(set,{color(v)})
5:     else
6:         for any child node u of v do
7:             union(set,ColorSetSize(u))
8:         end for
9:     end if
10:    C(v) ← size(set)
11:    return set
12: end function
13: ColorSetSize(root(T))
```

[23]). Their running time, due to a multiplicative inverse of the Ackermann function, is not linear, but, due to its simplicity and to the small constants involved, they turn out to be very fast, in practice. From a theoretical point of view, Tarjan et al. [21] showed how to get rid of the multiplicative Ackermann function when the the tree of the union is known in advance. In [46, 47, 48] it is shown that if the *finds* are performed in post order (at most one find for each element) then the overall amortized time is truly linear.

Recently, many algorithms appeared that, under some special conditions, perform *union-find* in linear time (see, for instance, [16, 30]) even including a new operation *delete* [4, 7, 14].

1.5.1 A New Color-Set-Size Algorithm

Definition Given a tree T with colored leaves, the Color-Set-Size (CSS) problem asks, for all internal node v, the number of different colors present in the sub tree rooted in v.

For the sake of easy, we firstly present a simple algorithm that solves the CSS problem in a non efficient way and we refine it later into an efficient linear solution.

Assume a tree T is given and any node in T is augmented by a counter C. We follow the simple idea of dynamically assign a set of colors to every node, where the colors are the unique colors present in the rooted subtree. Proceeding in a recursive way along a post order visit of the given tree, we assign to a leaf a set containing the color of the leaf, and we assign to an internal node the union of the sets coming from its children. At the end of any recursive step, we store the size of the current set to the color counter C of the examined node. Algorithm 0 follows this idea.

Algorithm 1 Recursive processing the tree T to compute the C values. *make-set*, *union* and *find* are common operations on *union-find* data structures. The operation *color* on a leaf v returns the associate color c_k.

```
1: function ColorSetSize(v)
2:     make-set(v)
3:     C(v) ← 0
4:     if v is a leaf then
5:         z ← find(P[color(v)])
6:         P[color(v)] ← v
7:         C(z) ← C(z) − 1
8:         C(v) ← 1
9:     else
10:        for any child node u of v do
11:            ColorSetSize(u)
12:            union(v,u)
13:            C(v) ← C(v) + C(u)
14:        end for
15:    end if
16: end function
17: ColorSetSize(root(T))
```

Algorithm 0 is conceptually very simple and it correctly computes, for any node v in T, the number of different colors C(v) present in the subtree rooted in v. A proof of this fact is straightforward and uses a simple property on sets. Unfortunately, Algorithm 0 cannot be implemented in efficient time, because of the union of sets which are not guaranteed to be disjoint each other. We can improve it by keeping sets of sibling nodes disjoint. We maintain sets of leaves instead of set of colors and we use, in a book-keeping strategy, a global array of previous occurrence of colors to maintain sets whose colors are disjoint. We also keep track of the size of the set associated to a node by maintaining this value in the color counter field of the node. Hence, we initialize the C counter of any node to 0, and we increment it accordingly to the cardinality of the set associated to it. In order to manage the set of leaves, we use a classic *union-find* data structure for disjoint sets. Instead of associate a set to a node, we include such node into the set and we use such node as the representative of the set. Hence, for any node v, there exist a set having v as representative, which contains all the nodes in the subtree rooted in v. Algorithm 1 is the resulting algorithm.

Let us summarize Algorithm 1 steps in what follows. Assume that K is the number of different colors in the given tree T. The nodes in the tree are augmented with a color counter field C, which constitutes the algorithm output. An auxiliary global array P (P stands for "previous visited leaf") of size K is used along a recursive post-order traversal of T in order to keep track of the previous occurrence of color $c_k, 1 \leq k \leq K$.

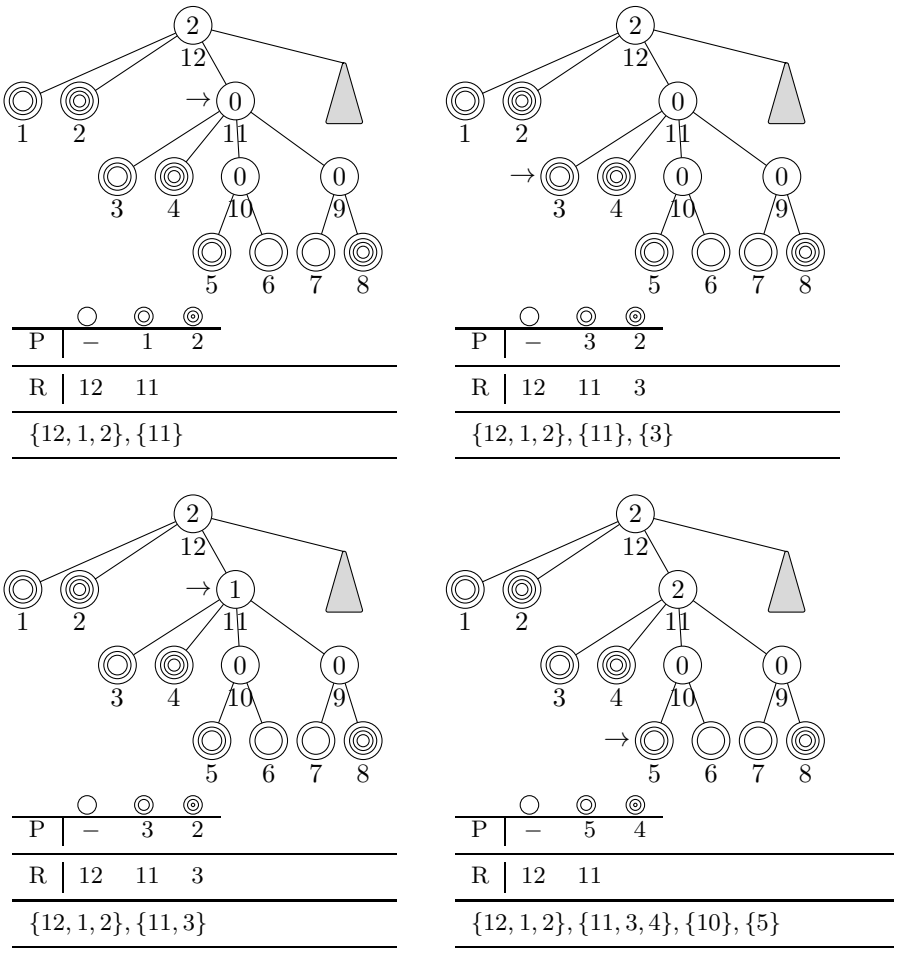


Figure 1.3 Some key steps of Algorithm 1 illustrated. Algorithm 1 computes C for any node v in the given tree. Array P is reported together with the list R of recursive calls of function ColorSetSize, and the sets maintained by the *union-find* data structure.

Recursively, we associate to any node a set containing itself and the union of the sets associated to its children and we add children color counter values to the parent counter. Once a leaf is visited, we find the root of the smallest subtree containing it and the previous visited leaf having the same color, and we decrement its color counter as this color will be added again at the end of the recursion of such subtree. At the end of each recursion, the color counter field C of the examined node is fixed, i.e. it does not change any more, and it stores the number of different colors in the rooted subtree. Figure 1.3 illustrates some steps of an example.

Proposition 1.5.1 *Assume a tree T is given. For all internal node v in T, ColorSetSize of v correctly computes the C value of any node in the subtree rooted in v, v included.*

Proof: The property that we want to prove is the following: After a call of function ColorSetSize(v) the set that has v as representative contains the node v and all the nodes in its rooted subtree T_v. Moreover, ColorSetSize(v) correctly computes the C value of all the nodes in the subtree T_v.

The proof is by induction on the number of nodes (internal nodes and leaves) in the subtree T_v rooted in v.

If the number of nodes in T_v is 1, its root v is a leaf and the base of the induction is clearly correct. Indeed, C(v), i.e. the color counter of node v, is set to 1 on line 8. Notice that the global array P is initialized with *nill* pointers. Let $u_1 \cdots u_h$, $h \geq 1$, be the sequence of children of v in the same order as they are considered in the **for** cycle at line 10. The inductive hypothesis holds on the tree $T_{v|u_h}$ that is the original subtree T_v pruned by the subtree rooted in u_h, and it holds also for T_{u_h} itself. Let us focus on the execution of function ColorSetSize(v) and freeze it after that the child u_{h-1} has been processed by the whole **for** cycle on lines 10–14. Up to this point, ColorSetSize(v) has executed the same operations as ColorSetSize(root($T_{v|u_h}$)) would have executed. We call C'(v) the value of C(v) at this step of the algorithm. Hence, by inductive hypothesis applied to $T_{v|u_h}$, the set which has v as representative contains the node v, the internal nodes in its subtree, and all the leaves that are present in $T_{v|u_h}$. Moreover, the C values on all nodes of the subtree $T_{v|u_h}$ have been correctly computed and have been stored in their C fields, C'(v) included. Notice that, all the leaves in subtree $T_{v|u_h}$ are contained in the set having v as representative by effect of unions at line 12.

Let us move forward on the execution of ColorSetSize(v) considering **for** cycle on line 10 for the child u_h. Let us also suppose that the number of leaves in $T_{v|u_h}$ having a color that also appears in T_{u_h} is d, that is, the number of duplicated colors in those two subtrees. After that we make the call ColorSetSize(u_h) on line 11, by inductive hypothesis applied to T_{u_h}, the set which has u_h as representative contains the node u_h and all the nodes in T_{u_h}, and the C values on all nodes in T_{u_h}, C(u_h) included, have been correctly computed. In the meantime, for each leaf in T_{u_h}, the color counter of the representative of the set containing the previous occurrence of a leaf having the same color (pointed by array P) has been decremented by effect of lines 5 and 7, and the array P is then accordingly updated at line 6. Hence, since d is the number of colors in $T_{v|u_h}$ that appear also in T_{u_h} and since all the leaves in $T_{v|u_h}$ are, at this step, contained in the v-represented set, the color counter of node v is now equal to C(v) = C'(v) $- d$.

After that, at lines 13, the number of color in T_{u_h}, i.e. C(u_h) $\geq d$ which is correct by induction hypothesis, is added to C(v) without counting duplicate colors, and the set having v as representative contains, v itself, the nodes in

426 $T_{v|u_h}$ by hypothesis and the nodes in T_{u_h} by the union performed at line 12.
427 This concludes the proof. ■

428 Let us now analyze the time and space requirements of Algorithm 1. From
429 a simple analysis of the pseudocode of Algorithm 1, it is easy to see that
430 any recursion of Algorithm 1 execute some constant time assignments and
431 at most one of each make-set, union and find operations of the *union-find*
432 data structure. Let us call t_m, t_u, t_f the time of make-set, union and find
433 operations, respectively.

434 **Theorem 1.5.2** *Assume tree T having n nodes is given. Algorithm* 1 *runs*
435 *in $O(n\ (t_m + t_u + t_f))$ time.*

436 The make-set and union operations are standard constant time operation
437 in many common linear space *union-find* data structure (see, for instance,
438 [23, 21, 47, 46, 48, 22]). There are many classical *union-find* solution provid-
439 ing a find operation in $O(\log m)$ time, where m is the cardinality of the set
440 containing the queried element [23]. The amortized time used by such struc-
441 tures to execute a find query over the n elements is $O(\alpha(m))$, where α is a
442 functional inverse of Ackermanns function, that is an extremely slow growing
443 function which, for any practical number m, is less than 4. Practically speak-
444 ing, $\alpha(m)$ is usually considered just a small constant. Moreover, since there
445 exist *union-find* implementations that are very fast in practice, such solutions
446 are usually preferred to theoretically more efficient solutions.

447 **Corollary 1.5.3** *Assume tree T having n nodes is given. By using any data*
448 *structure for disjoint set in [23] using* quick *union, Algorithm* 1 *runs in amor-*
449 *tized $O(n\ \alpha(n))$ time and linear space in the RAM model.*

450 However, the *union-find* data structure presented in [21] answer to find
451 queries in amortized linear time, when a find operation is followed by a union
452 operation and the sequence of find operations is known in advance. This is the
453 case of Algorithm 1, where the union operations strictly follow the recursive
454 visit of the input tree.

455 **Corollary 1.5.4** *Assume tree T having n nodes is given. By using Tarjan's*
456 *data structure [21], Algorithm* 1 *runs in amortized linear time and linear space*
457 *in the RAM model.*

458 1.6 REDUCING SPACE

459 In this section we describe how to reduce the space used by the Color-Set-Size
460 solution presented in previous section and the total space used for the Longest
461 Common Substring problem.
462 A first observation leads to keep low the space used by the *union-find* data
463 structure.

Algorithm 2 Recursive processing the tree T to compute the C values. *make-set*, *union*, *find* and *delete* are common operations on *union-find-delete* data structures.

```
1: function ColorSetSize(v)
2:     make-set(v)
3:     C(v) ← 0
4:     if v is a leaf then
5:         z ← find(P[color(v)])
6:         delete(P[color(v)])
7:         P[color(v)] ← v
8:         C(z) ← C(z) −1
9:         C(v) ← 1
10:    else
11:        for any child node u of v do
12:            ColorSetSize(u)
13:            union(v,u)
14:            C(v) ← C(v) + C(u)
15:            if u is NOT a leaf then
16:                delete(u)
17:            end if
18:        end for
19:    end if
20: end function
21: ColorSetSize(root(T))
```

Algorithm 1 uses, at any moment, the leaves pointed by array P and no other leaves are used in order to adjust the color counters in case of duplicates. At any time, those leaves are at most K, same as the number of encountered different colors. Moreover, the internal nodes are to be released as soon as their subtree is entirely visited. Putting these observation together, we adapt previous algorithm to use the delete operation provided by Alstrup et. al [4] to keep low the space used by the *union-find-delete* data structure. Algorithm 2 reports the adapted pseudocode.

The delete operation is not part of the classical set of operation of the *union-find* data structures, where the element in the set are always maintained once they are created by a make-set operation. The delete operation appears only in an extension of the *union-find* data structures called *union-find-delete* data structures [4, 7]. Recall that, in [4], the delete operation is also a constant-time operation, same as make-set and union operations, while a find takes $O(\alpha(m))$ amortized time, where m is the number of elements in the set returned by the find operation, and α is a functional inverse of Ackermanns function. The space is linear in the number of elements simultaneously maintained, at any time. Algorithm 2 maintains, at any time, one internal node for any nested call to the function ColorSetSize and, at most, K leaves.

The number of recursion of function ColorSetSize is bounded by the height of the tree T. Then, for a general tree having n nodes, the number of recursion is n in the worst case, but it is expected to be much less in practical cases, where the tree T is more balanced.

When CSS is used for the LCS problem, the tree T is a generalized suffix tree and it is proved by W. Szpankoski in [62] that the height of such trees tends almost surely to $O(\log n)$, where n is the length of the text. This result holds also for biological collections [63].

Moreover, a space linear in the height of the tree is also implicitly used by the recursion stack. Even if algorithms in this chapter are state in recursive form, it is easy to obtain non recursive versions by using standard programming techniques, where a stack is explicitly used to perform the post order visit of the tree.

Let us summarize the results presented so far in the following proposition.

Proposition 1.6.1 *Assume a tree T is given having n nodes and colored leaves with K different colors. By using the* union-find-delete *data structure of [4], Algorithm* 2 *runs in $O(n\ \alpha(K))$ amortized time and uses $O(K + height(T))$ extra space to compute all the C values.*

The second observation of this section concerns the output size of CSS and LCS problems. Even if the two problems are deeply related, their outputs are totally different in size. In fact, the problem CSS has input and output of the size of the given tree T, while LCS has output of size K, that is the number of different colors.

While the output of the first problem maintains the tree structure, the output of the second can be an array L that, for each k, $2 \leq k \leq K$, contains the length of a longest substring that appears in at least k strings *and*, optionally, a pointer to one of such substrings (i.e., the document and the position where it appears). The outputs is then proportional to K (more precisely it has the size of $\mathrm{O}(K)$ integers).

We want to obtain a direct solution of the LCS problem that does not use a full solution of the CSS problem and, consequently, does not use CSS output space. First of all, we notice that array L can be obtained during the tree traversal of above algorithms. Indeed, if a node has been examined together its rooted subtree during the visit, that is, at the end of a recursion, its color counter C will not change anymore. Therefore, if the color counter is equal to k and the string depth of the node is greater than the length contained in $\mathrm{L}(k)$, then we replace the contents of $\mathrm{L}(k)$ with the string length of such node[1]. In the meantime, a pointer to one occurrence (document and starting position) of such string can be stored as well.

In this way, $\mathrm{L}(k)$ contains, at any moment, the length of a longest string that is common to exactly k substrings, up to that step of the visit over the

[1]The string length of a node is classically defined as the length of the concatenated labels on the path from the root to such node.

tree. Along a further linear scan of the array L, from the smallest to the bigger index, we can simply adjust the values to fit the at least k requirement of the LCS problem.

Function ColorSetSize of Algorithm 2 can easily reports the correct C value of a node as soon as a recursion ends. But, the algorithm also stores temporary values in the color counter C of parent nodes, while a subtree is traversed. The number of subtree roots whose color counter can virtually be decremented, according to the presence of duplicate colors, is, at most, height(T). Hence, we maintain such temporary counters in a global array C of length height(T) indexed by node depth.

Now, since we already know how to get rid of the C counter fields in the tree nodes, we notice that it is possible to use a compressed index to simulate the functionalities of a suffix tree, and, in particular, to perform a post order visit on it. Obviously, such compressed structures have query time and space requirements strongly dependent on the underling Compressed Suffix Array used and some parameters.

The research of efficient (mainly in space) data structures that can be used instead of suffix trees has become an independent research field. Up to less than a decade ago, the most commonly used data structures are suffix trees, suffix arrays, DAWGs and Compact DAWGs. Usually any problem that can be settled by the aid of one of such data structure can also be settled by using any of the other ones. Despite this fact, the passage from one data structure to another is not automatic nor always easy. Each of these structures has some advantage and some disadvantage. Some relation among the data structures and their size is reported in [11]. The size of an implementation of the above data structures is often evaluated by the average number of bytes necessary to store one letter of the original text. It is commonly admitted that these ratios are 4 for suffix arrays, 9 to 11 for suffix trees, and 5 for CDAWGs (cf. [11] for further information).

In the meantime, other space-efficient related data structures started to appear in research papers. For instance, Compressed Suffix Array and Compressed Suffix Tree are showed to have the potential to replace in many applications suffix trees by using less space exploiting redundancy of the text. [1, 2, 19, 20, 24, 26, 37, 38, 45, 53, 55, 58].

Latter data structures are space thrifty by using only $nH_k + O(n)$ *bits*. We refer to Fischer et al paper [20] for a comparison between different trade-off between occupied space and query time for some of these compressed indexes, where they summarize crucial values in [20, Table 1]. We propose Algorithm 3 as a variant of our LCS solution using a Compressed Suffix Tree. Given a document collection $\mathcal{K}$, we associate a unique color to each of the K documents in $\mathcal{K}$. The Algorithm returns the array L and S; L contains, for $1 \leq k \leq K$, the length L[k] of a longest substring common to at least k documents, and S[k] contains a reference to one of such substrings. Function color retrieves the color associated to a leaf. Function make-set, union, find and delete are standard operations on *union-find-delete* data structures. Function

Algorithm 3 Processing the collection $\mathcal{K}$ to solve the LCS problem by using a *compressed suffix tree* and a *union-find* data structure.

```
 1: function LightweightCSS(v)
 2:     make-set(v)
 3:     C[node-depth(v)] ← 0
 4:     if v is a leaf then
 5:         z ← find(P[color(v)])
 6:         delete(P[color(v)])
 7:         P[color(v)] ← v
 8:         C[node-depth(z)] ← C[node-depth(z)] − 1
 9:         C[node-depth(v)] ← 1
10:     else
11:         for any child node u of v do
12:             C[node-depth(v)] ← C[node-depth(v)] + LightweightCSS(u)
13:             union(v,u)
14:             if u is NOT a leaf then
15:                 delete(u)
16:             end if
17:         end for
18:     end if
19:     if L[C[node-depth(v)]] < string-depth(v) then
20:         L[C[node-depth(v)]] ← string-depth(v)
21:         S[C[node-depth(v)]] ← pos(v)
22:     end if
23:     return C[node-depth(v)]
24: end function
25:
26: Initialize P, L, and S
27: Build CST_K
28: LightweightCSS(root(CST_K))
29: for i = K, K − 1, ... 2 do
30:     if L[i − 1] < L[i] then
31:         L[i − 1] ← L[i]
32:         S[i − 1] ← S[i]
33:     end if
34: end for
35: return L,S
```

node-depth, string-depth and pos are standard functions on suffix trees (and compressed suffix trees), as well as, to know if a node is a leaf, and performing a post order traversal of a rooted subtree. Notice that, every call to function LightweightCSS needs to query the compressed suffix tree to retrieve children, node-depth and string-depth of a given node. In all the known compressed index at least one of such queries takes logarithmic time.

Proposition 1.6.2 *Assume a collection $\mathcal{K}$ of K documents having total length n is given. By using the Compressed Suffix Tree $CST_{\mathcal{K}}$ of [59] and the* union-find-delete *data structure of [4], it is possible to solve the LCS problem in $O(n\ \alpha(K)\log^{\epsilon} n)$ amortized time and $O(n + K + height(CST_{\mathcal{K}})) + nH_k$ bits space, where $0 < \epsilon < 1$.*

As last observation, we notice that in many compressed text index based on a compressed suffix array, many functionality are obtained by using a Range Minimum Query on the Longest Common Prefix (LCP) table, and, moreover, they support LCA queries without using extra space or extra preprocessing.

Let us recall how simulate a LCA query over a suffix array. Given a tree T and two nodes u and v, the LCA problem wants to find the lowest node in the tree that has both u and v in its rooted subtree. This problem was posed in [3], but the first linear preprocessing solution and constant time query is due to Harel and Tarjan [31] based on heavy path decomposition. Many improvements, mostly in practical space requirements, appeared recently (see, for instance, [8, 17, 60]) which use different approaches: Cartesian trees, geometrical range queries and lookup tables.

In the case where T is the suffix tree $ST_{\mathcal{T}}$ of a text $\mathcal{T}$, the LCA of two leaves is equivalent to the Longest Common Prefix of two suffixes. By using the Suffix Array SA of $\mathcal{T}$ and the associated LCP table (which contains the LCP values of consecutive suffixes in lexicographic order, that is, $\mathrm{LCP}[i] = \mathrm{LCP}(SA[i-1], SA[i])$), it is possible to find the LCP of two given suffix u and v by computing the minimum value in $\mathrm{LCP}[x\,..\,y]$, where x and y are the index of i and j in the SA, respectively. Therefore, suppose to have the inverse suffix array SA^{-1} ($SA^{-1}[i] = x \iff SA[x] = i$), it is possible to compute in constant time the LCP of i and j as $min\ \{LCP[x] \mid SA^{-1}[i] \leq x \leq SA^{-1}[j]\}$ by using the range minimum query algorithm (see for instance [18]).

Since many CSTs [59, 58, 18] support LCA query using no extra space, we can now remove the *union-find* data structure from our solution and use LCA queries instead of it. We simply replace a find(u) call in Algorithm 3 by a $\mathrm{LCA}(u, v)$ query. In fact, due to the post order visit of the tree, i.e., the nested recursion of our algorithms, those two operations are equivalent. The find(u) query returns the representative of the set containing the given node, and, by construction, it is the root of the smallest subtree containing u and v, that is exactly $\mathrm{LCA}(u, v)$. Algorithm 4 reflects this observation.

We state our results by using the most efficient compressed index, to the best of our knowledge, in terms of space occupancy.

Proposition 1.6.3 *Assume a collection $\mathcal{K}$ of K documents, $K \geq 2$, having total length n is given. By using the Compressed Suffix Tree $CST_{\mathcal{K}}$ in [58], Algorithm 4 solves the LCS problem in $O(n\ \log^{1+\epsilon} n)$ time and $O(K + height(ST_{\mathcal{T}})) + nH_k + o(n)$ bits of space, where H_k is the k-order empirical entropy of $\mathcal{T}$, and $0 < \epsilon < 1$.*

Algorithm 4 Processing the collection $\mathcal{K}$ to solve the LCS problem by using a *compressed suffix tree* supporting the LCA query.

```
1: function LightweightCSS(v)
2:     C[node-depth(v)] ← 0
3:     if v is a leaf then
4:         z ← LCA(P[color(v)])
5:         P[color(v)] ← v
6:         C[node-depth(z)] ← C[node-depth(z)] − 1
7:         C[node-depth(v)] ← 1
8:     else
9:         for any child node u of v do
10:            C[node-depth(v)] ← C[node-depth(v)] + LightweightCSS(u)
11:        end for
12:    end if
13:    if L[C[node-depth(v)]] < string-depth(v) then
14:        L[C[node-depth(v)]] ← string-depth(v)
15:        S[C[node-depth(v)]] ← pos(v)
16:    end if
17:    return C[node-depth(v)]
18: end function
19:
20: Initialize P, L, and S
21: Build CST_K
22: LightweightCSS(root(CST_K))
23: for i = K, K − 1, ... 2 do
24:    if L[i − 1] < L[i] then
25:        L[i − 1] ← L[i]
26:        S[i − 1] ← S[i]
27:    end if
28: end for
29: return L,S
```

Notice that, $O(K + \text{height}(ST_{\mathcal{T}})) + nH_k + o(n)$ bits can be sublinear in n. Recall that, given a collection of strings of total length n whose longest string has length m, the height of the generalize suffix tree of such collection is, in the worst case, $O(m)$, and, in average, $O(\log m)$, even for biological sequences [62, 63]. Moreover, due to the simplicity of this solution, which essentially is a post order visit on a Compressed Suffix Tree supporting LCA queries and a simple book-keeping of K values, any further improvement of Compressed Suffix Tree in terms of query time and occupied space, can be directly integrated to above algorithms and leads to better performances.

References

1. A. Abeliuk, R. Cánovas, and G. Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013.
2. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.
3. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. *SIAM J. Comput.*, 5(1):115–132, 1976.
4. S. Alstrup, I. L. Gørtz, T. Rauhe, M. Thorup, and U. Zwick. Union-Find with constant time deletions. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 78–89. Springer, 2005.
5. A. Apostolico. *Combinatorial Algorithms on Words*, volume 12, chapter The Myriad Virtues of Subword Trees, pages 85–96. Springer Berlin Heidelberg, 1985.
6. M. Arnold and E. Ohlebusch. Linear time algorithms for generalizations of the longest common substring problem. *Algorithmica*, 60(4):806–818, 2011.
7. A. M. Ben-Amram and S. Yoffe. A simple and efficient Union-Find-Delete algorithm. *Theor. Comput. Sci.*, 412(4-5):487–492, 2011.
8. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In Gonnet et al. [25], pages 88–94.
9. O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

10. D. Breslauer and G. F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013.
11. M. Crochemore. Reducing space for index implementation. *Theoret. Comput. Sci.*, 292(1):185–197, 2003.
12. M. Crochemore, A. Gabriele, F. Mignosi, and M. Pesaresi. On the longest common factor problem. In G. Ausiello, J. Karhumäki, G. Mauri, and C.-H. L. Ong, editors, *IFIP TCS*, volume 273 of *IFIP*, pages 143–155. Springer, 2008.
13. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. 392 pages.
14. M. B. Dillencourt, H. Samet, and M. Tamminen. A general approach to connected-component labelling for arbitrary image representations. *J. ACM*, 39(2):253–280, 1992.
15. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, Nov. 2000.
16. C. Fiorio and J. Gustedt. Two linear time Union-Find strategies for image processing. *Theor. Comput. Sci.*, 154(2):165–181, 1996.
17. J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In M. Lewenstein and G. Valiente, editors, *CPM*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006.
18. J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In B. Chen, M. Paterson, and G. Zhang, editors, *ESCAPE*, volume 4614 of *Lecture Notes in Computer Science*, pages 459–470. Springer, 2007.
19. J. Fischer and V. Heun. Range median of minima queries, super-cartesian trees, and text indexing. In M. Miller and K. Wada, editors, *IWOCA*, pages 239–252. College Publications, 2008.
20. J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.
21. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985.
22. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985.
23. Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344, 1991.
24. S. Gog and E. Ohlebusch. Compressed suffix trees: Efficient computation and storage of lcp-values. *ACM Journal of Experimental Algorithmics*, 18, 2013.
25. G. H. Gonnet, D. Panario, and A. Viola, editors. *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*. Springer, 2000.
26. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

692 27. M. Gundry, W. Li, S. B. Maqbool, and J. Vijg. Direct, genome-wide assessment
693 of DNA mutations in single cells. *Nucleic Acids Res.*, 40(5):2032–2040, 2012.

694 28. D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science
695 and Computational Biology*. Cambridge University Press, 1997.

696 29. D. Gusfield, G. M. Landau, and B. Schieber. An efficient algorithm for the all
697 pairs suffix-prefix problem. *Inf. Process. Lett.*, 41(4):181–185, 1992.

698 30. J. Gustedt. Efficient Union-Find for planar graphs and other sparse graph
699 classes. *Theor. Comput. Sci.*, 203(1):123–141, 1998.

700 31. D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors.
701 *SIAM J. Comput.*, 13:338–355, 1984.

702 32. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ances-
703 tors. *SIAM J. Comput.*, 13(2):338–355, 1984.

704 33. M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient multiple genome alignment. In
705 *ISMB*, pages 312–320, 2002.

706 34. L. C. K. Hui. Color set size problem with application to string matching. In
707 A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *CPM*, volume
708 644 of *Lecture Notes in Computer Science*, pages 230–243. Springer, 1992.

709 35. L. C. K. Hui. A practical algorithm to find longest common substring in linear
710 time. *Int. J. of Comput. Syst. Sci. & Eng.*, 15:73–76, 2000.

711 36. J. Kärkkäinen, P. Sanders, and S. Burkhardt. Simple linear work suffix array
712 construction. *J. ACM*, 53(6):918–936, 2006.

713 37. D. K. Kim, M. Kim, and H. Park. Linearized suffix tree: an efficient index data
714 structure with the capabilities of suffix trees and suffix arrays. *Algorithmica*,
715 52(3):350–377, 2008.

716 38. D. K. Kim and H. Park. A new compressed suffix tree supporting fast search
717 and its construction algorithm using optimal working space. In A. Apostolico,
718 M. Crochemore, and K. Park, editors, *CPM*, volume 3537 of *Lecture Notes in
719 Computer Science*, pages 33–44. Springer, 2005.

720 39. D. K. Kim, J. S. Sim, H. Park, and K. Park. Constructing suffix arrays in linear
721 time. *J. Discrete Algorithms*, 3(2-4):126–142, 2005.

722 40. K. P. Kim, S. S. Cho, K. K. Lee, M. H. Youn, and S.-T. Kwon. Improved ther-
723 mostability and PCR efficiency of Thermococcus celericrescens DNA polymerase
724 via site-directed mutagenesis. *Journal of Biotechnology*, 155(10):156–163, 2011.

725 41. D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings.
726 *SIAM J. Comput.*, 6(2):323–350, 1977.

727 42. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J.
728 Discrete Algorithms*, 3(2-4):143–156, 2005.

729 43. I. Lee, C. S. Iliopoulos, and K. Park. Linear time algorithm for the longest
730 common repeat problem. *J. Discrete Algorithms*, 5(2):243–249, 2007.

731 44. I. Lee and Y. J. Pinzon. A simple algorithm for finding exact common repeats.
732 *IEICE Transactions*, 90-D(12):2096–2099, 2007.

733 45. J. Lin, Y. Jiang, and D. A. Adjeroh. The virtual suffix tree. *Int. J. Found.
734 Comput. Sci.*, 20(6):1109–1133, 2009.

735 46. M. Loebl and J. Nesetril. Linearity and unprovability of set union problem
736 strategies. In J. Simon, editor, *STOC*, pages 360–366. ACM, 1988.

737 47. M. Loebl and J. Nesetril. Postorder hierarchy for path compressions and set
738 union. In J. Dassow and J. Kelemen, editors, *IMYCS*, volume 381 of *Lecture*
739 *Notes in Computer Science*, pages 146–151. Springer, 1988.

740 48. J. M. Lucas. Postorder disjoint set union is linear. *SIAM J. Comput.*, 19(5):868–
741 882, 1990.

742 49. W. Ludwig and K. Schleifer. Bacterial phylogeny based on 16s and 23s rRNA
743 sequence analysis. *FEMS Microbiology Reviews*, 15(23):155 – 173, 1994.

744 50. M. G. Maaß. Matching statistics: efficient computation and a new practical
745 algorithm for the multiple common substring problem. *Softw., Pract. Exper.*,
746 36(3):305–331, 2006.

747 51. U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string
748 searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

749 52. E. M. McCreight. A space-economical suffix tree construction algorithm. *J.*
750 *ACM*, 23(2):262–272, 1976.

751 53. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput.*
752 *Surv.*, 39(1), 2007.

753 54. E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rear-*
754 *rangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

755 55. E. Ohlebusch and S. Gog. A compressed enhanced suffix array supporting fast
756 string matching. In J. Karlgren, J. Tarhio, and H. Hyyrö, editors, *SPIRE*,
757 volume 5721 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 2009.

758 56. S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array con-
759 struction algorithms. *ACM Comput. Surv.*, 39(2), 2007.

760 57. T. Roychowdhury, A. Vishnoi, and A. Bhattacharya. Next-generation Anchor
761 Based Phylogeny (NexABP): Constructing phylogeny from Next-Generation Se-
762 quencing data. *Scientific reports*, 3, 2013.

763 58. L. M. S. Russo, G. Navarro, and A. L. Oliveira. Fully compressed suffix trees.
764 *ACM Transactions on Algorithms*, 7(4):53, 2011.

765 59. K. Sadakane. Compressed suffix trees with full functionality. *Theor. Comp.*
766 *Sys.*, 41(4):589–607, Dec. 2007.

767 60. B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification
768 and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

769 61. B. Snel, P. Bork, and M. A. Huynen. Genome phylogeny based on gene content.
770 *Nature genetics*, 21(1):108–110, 1999.

771 62. W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic
772 behaviors. *SIAM J. Computing*, 22:1176–1198, 1996.

773 63. W. Szpankowski. *Average Case Analysis of Algorithms on Sequences (Wiley*
774 *Series in Discrete Mathematics and Optimization)*. Wiley-Interscience, 2001.

775 64. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260,
776 1995.

777 65. P. Weiner. Linear pattern matching algorithms. In *SWAT (FOCS)*, pages 1–11.
778 IEEE Computer Society, 1973.

779 66. C. R. Woese and G. E. Fox. The concept of cellular evolution. *Journal of*
780 *molecular evolution*, 10(1):1–6, 1977.